

# A Backward Algorithm for the Multiprocessor Online Feasibility of Sporadic Tasks\*

Gilles Geeraerts and Joël Goossens and Thi-Van-Anh Nguyen

Université libre de Bruxelles (ULB), Faculté des Sciences,  
Département d'Informatique, Belgium  
gigeerae@ulb.ac.be  
joel.goossens@ulb.ac.be  
thi-van-anh.nguyen@ulb.ac.be

## Abstract

The online feasibility problem (for a set of sporadic tasks) asks whether there is a scheduler that always prevents deadline misses (if any), whatever the sequence of job releases, which is *a priori* unknown to the scheduler. In the multiprocessor setting, this problem is notoriously difficult. The only exact test for this problem has been proposed by Bonifaci and Marchetti-Spaccamela: it consists in modelling all the possible behaviours of the scheduler and of the tasks as a graph; and to interpret this graph as a game between the tasks and the scheduler, which are seen as antagonistic players. Then, computing a correct scheduler is equivalent to finding a winning strategy for the ‘scheduler player’, whose objective in the game is to avoid deadline misses. In practice, however this approach is limited by the intractable size of the graph. In this work, we consider the classical *attractor* algorithm to solve such games, and introduce *antichain techniques* to optimise its performance in practice and overcome the huge size of the game graph. These techniques are inspired from results from the formal methods community, and exploit the specific structure of the feasibility problem. We demonstrate empirically that our approach allows to dramatically improve the performance of the game solving algorithm.

## 1 Introduction

The model of *sporadic tasks* is nowadays a well-established model for real-time systems. In this model, one considers a set of real-time *tasks*, that regularly release *jobs*. The jobs are the actual computational payloads, and the *scheduler* must assign those jobs to the available processor(s) of the platform in a way that ensures no job misses its *deadline* (i.e., ensuring that all jobs are granted enough computation time within a given time frame). The term *sporadic* refers to the fact that there is some uncertainty on the moment at which tasks release jobs:

---

\*This work has been supported by the F.R.S./FNRS PDR grant FORESt: Formal verification techniques for REal-time Scheduling problems.

in the sporadic model, an *inter-arrival time*  $T_i$  is associated with each task  $\tau_i$ . This means that at least  $T_i$  time units will elapse in-between two job releases of the task  $\tau_i$ , but the release can be delayed by an arbitrary long time (this is in contrast with the Liu and Layland model [LL73] where the tasks release jobs *periodically*, i.e., every  $T_i$  time unit).

In the present paper, we consider the so-called *online feasibility* problem for such *sporadic* tasks on a *multiprocessor platform*, i.e., a platform that boasts several identical computational units on which the scheduler can choose to run any active job. The *online feasibility problem* asks, given a set of (sporadic) tasks to *compute*—if possible—a scheduling policy for those tasks ensuring that no task ever misses its deadline. An important assumption is that the scheduler is *not clairvoyant*, i.e., it has no information on the future job releases, apart from the information that can be deduced from the current history of execution. Hence, the scheduler that has to be computed must be able to react to any potential sequence of job releases (it is thus an *online scheduler*<sup>1</sup>).

This feasibility problem of sporadic tasks on multiprocessor platforms is appealing both from the practical and the theoretical point of view. On the practical side, the problem is rich enough to model realistic situations. On the theoretical side, the different degrees of non-determinism of the model (the exact release times of the jobs, the exact execution duration of jobs, the many possible decisions of the scheduler when there are more active tasks than available CPUs) make its behaviour difficult to predict.

Indeed, while several necessary and sufficient conditions have been identified for feasibility, these criteria are not able to decide feasibility of all systems. Moreover, while a collection of optimal schedulers for *uniprocessors* is known, *optimal* online multiprocessor scheduling of sporadic real-time tasks is known to be impossible [FGB10], at least for constrained or arbitrary deadline systems.

In this paper, we build on the ideas introduced by Bonifaci and Marchetti-Spaccamela [BM10], and consider a *game-based model* for our feasibility problem. Intuitively, we regard the scheduler as a player that competes against the coalition of the tasks. The moves of the scheduler are the possible assignments of active jobs to the CPUs, and the moves of the tasks are the possible job releases. The objective of the scheduler in this game is to avoid deadline misses, and we assume that the tasks have the opposite objective. This models the fact that the tasks will not collaborate (i.e., release jobs in a way that avoids deadline misses) with the scheduler. Then, it should be clear that the system of tasks is *feasible* iff there is a *winning strategy* for the scheduler (one that enforces the scheduler's objective whatever the tasks play). When the system is feasible, a scheduler can be extracted from this winning strategy. To the best of our knowledge, this is the only *exact* feasibility test that has been proposed so far for this problem.

Concretely, the game we consider is a special case of *game played on graph*, that we call *scheduling games*: the nodes of the graph model the possible states of the system, and the edges model the possible moves of the players. This approach is akin to the *controller synthesis problem* which has been extensively considered in the formal methods community for the past 25 years [PR89]: given a model of a computer system interacting with an environment and given a property that this system must satisfy (typically, a safety property), the controller synthesis

---

<sup>1</sup>Note that our approach requires a significant offline preprocessing phase, hence the schedulers that are produced can be characterised as *semionline*, according to some authors.

problem consists in computing a *controller* that enforces the property, whatever the environment does. The parallel with our feasibility problem should be clear: the scheduler corresponds to the controller, the environment to the tasks, and the property is to avoid deadline misses. In the formal methods literature, the classical approach to the synthesis problem boils down to computing a winning strategy in a game.

The main practical limitation to this approach is the intractable size of the game graph—the so-called ‘state explosion problem’, a typical limitation of exhaustive verification (e.g. model-checking) and synthesis techniques that consider all possible system states. In our case, this graph is exponential in the size of the problem, so, building the whole graph, then analysing it (as proposed by Bonifaci and Marchetti-Spaccamela [BM10]) is infeasible in practice, even though the algorithms that compute winning strategies for games played on graphs are efficient (they run in polynomial time wrt the size of the graph). Nevertheless, the community of formal methods has proposed several techniques (such as efficient data structures, heuristics to avoid the exploration of all states, *etc.*...) that do overcome this barrier in practice (see for example [BCM<sup>+</sup>92]). Building on these results, we have shown, in previous works, that the same techniques have the potential to dramatically improve the running time of some graph-based algorithms solving real-time scheduling problems. More precisely, in [GGL13] and [GGNS] we have applied so-called *antichain* [DR10] techniques (introduced originally a.o. to speed up model checking algorithms) and demonstrated experimentally their practical interest.

**Contributions** In this paper, we continue our line of research and apply the antichain approach to improve the efficiency of the classical *attractor* (or *backward induction*) algorithm to solve scheduling games. At the core of our optimisation is a so-called *simulation relation* that allows one to avoid exploring a significant portion of the game’s nodes. While the antichain approach is a generic one, the definition of the simulation relation has to be tailored to the specific class of problem under consideration. Our simulation relation exploits tightly the peculiar structure of scheduling games. We perform a series of experiments (see Section 5) to demonstrate the usefulness of our new approach. Our enhanced algorithm performs one order of magnitude better than the un-optimised attractor algorithm (see Figure 4). Moreover, we manage to compute schedulers for systems that are not schedulable under the classical EDF scheduler: in particular, when the load of the system is close to 100%, our approach outperforms vastly EDF (see Figure 3).

**Related works** The feasibility problem for sporadic tasks is a well-studied problem in the real-time scheduling community. Apart from the very particular case where for each task the deadline corresponds to the period (implicit deadline systems), where polynomial time feasibility tests exist, it exhibits a high complexity. For instance, even in the uniprocessor case, the feasibility problem of recurrent (periodic or sporadic) constrained deadline tasks is strongly coNP complete [EY15]. For constrained/arbitrary deadline and sporadic task systems, several *necessary* or *sufficient* conditions have been identified for feasibility. However, these criteria are not sufficient to decide feasibility for all systems: there are some systems that do not meet any sufficient conditions (hence, are not surely feasible) and respect all identified necessary conditions (hence are not surely infeasible). We refer the reader to a survey by Davis and Burns [DB11].

As already mentioned, the only *exact* test (i.e., algorithm to decide the

	$C_i$	$D_i$	$T_i$
$\tau_1$	1	1	2
$\tau_2$	2	2	2
$\tau_3$	1	4	2

Table 1: The task set used as our running example

problem) is the one of Bonifaci and Marchetti-Spaccamela [BM10] that consists in reducing the problem to the computation of a winning strategy in a game. However no optimisations are proposed to make this approach practical. In a previous work [GGNS], we have already presented a game solving algorithm, enhanced by antichains, to solve the feasibility problem. Compared to the present paper, [GGNS] relies on the same model (Section 2) and the same partial order  $\sqsubseteq$ . However, the algorithm we consider here (Algorithm 1) is completely different. Its improvement based on antichains (Algorithm 2, discussed from Section 3.2 onward) is non-trivial and constitutes original work. Our experimental evaluation (Section 5) compares our two approaches and show that they are actually complementary.

## 2 Feasibility as a game

Let us now define precisely the *online feasibility problem of sporadic tasks* on a *multiprocessor platform*. Remember that *online* means that the scheduler has no a priori knowledge of the behaviour of the tasks; but must react, during the execution of the systems to the requests made by the tasks. To formalise this problem, we consider a set  $\tau = \{\tau_1, \dots, \tau_n\}$  of sporadic tasks to be scheduled on  $m$  identical processors. Each task  $\tau_i$  is characterised by three non-negative integer parameters  $C_i, D_i, T_i$ . Parameter  $C_i$  is the *worst case execution time* (WCET), i.e. an upper bound on the number of CPU ticks that are needed to complete the job.  $D_i$  is the *deadline*, relative to each job arrival. Each job needs to have been assigned  $C_i$  units of CPU time before its deadline, lest it will *miss its deadline* (for example if some job of task  $\tau_i$  is released at instant  $t$ , it will need  $C_i$  units of CPU time before  $t + D_i$ ). Finally,  $T_i$  is the *minimal interarrival time*: at least  $T_i$  time units must elapse between two job releases of  $\tau_i$ . We assume that jobs can be preempted and can be freely migrated between CPUs, without incurring any delay. An example of system with three tasks is given in Table 1. Finally, we assume that each job requires its WCET. Since the case where all jobs consume their WCET is always a possible scenario, this assumption is without loss of generality. Indeed, if there is no scheduler when the tasks are restricted to consume their whole WCET, then there will be no scheduler in the case where the tasks also have the option to complete earlier; and, if jobs complete earlier at runtime, it is always possible to keep the CPU idle for the remaining allocated ticks.

In such a real-time system, tasks freely submit new jobs, respecting their minimal interarrival time. Then, the scheduler is responsible to allocate the jobs to the CPUs (ensuring that no job misses its deadline), a CPU tick occurs (note that we consider a discrete time model), and a new round begins with the

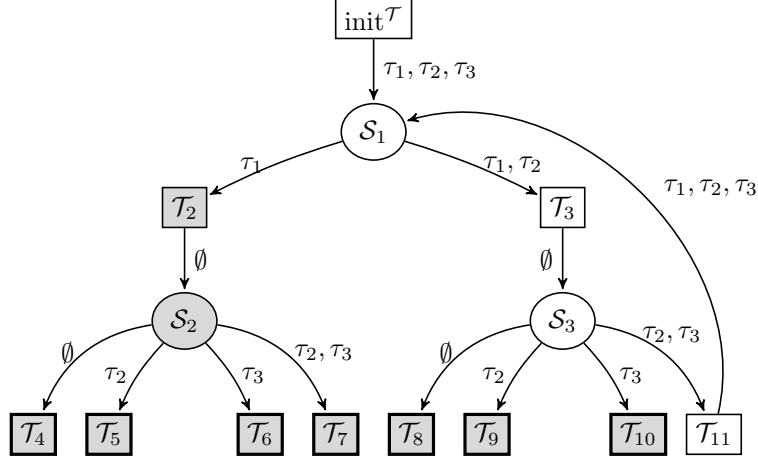


Figure 1: A prefix of the scheduling game corresponding to the system in Table 1. Square nodes belong to  $\mathcal{T}$  and the round ones to  $\mathcal{S}$ . Gray nodes are *losing*. Nodes with thick borders are *bad* nodes, i.e. where at least one task misses its deadline.

tasks submitting new jobs, etc. To model formally this semantics, we rely on the notion of *scheduling game*, as sketched in the introduction. Before defining formally this notion, let us introduce it through an example. Given a set of tasks  $\tau$  and a number of CPUs  $m$ , a scheduling game is played on a graph  $G_{\tau,m}$ , by two players which are the scheduler (called player  $\mathcal{S}$ ) and the coalition of the tasks (player  $\mathcal{T}$ ). In the case of the task set  $\tau$  of Table 1 upon a 2-CPU platform ( $m = 2$ ), a prefix of  $G_{\tau,m}$  is given in Figure 1. In this graph, the nodes model the system states and the edges model the possible moves of the players (all these notions will be defined precisely hereinafter). Each path is a possible execution of the system. The actual path which is played stems from the interaction of the players which play in turn: the square (respectively round) nodes ‘belong’ to  $\mathcal{T}$  ( $\mathcal{S}$ ), which means that  $\mathcal{T}$  ( $\mathcal{S}$ ) decides in those nodes what will be the successor node. The game starts in  $\text{init}^{\mathcal{T}}$ , where the tasks ( $\mathcal{T}$ ) decide which jobs to submit. In the full graph  $G_{\tau,m}$ , there are  $2^3 = 8$  different possibilities, but we display here only the case where all tasks ( $\tau_1$ ,  $\tau_2$  and  $\tau_3$ ) submit a job, which moves the system to state  $\mathcal{S}_1$ . In this state, the scheduler must decide which tasks to schedule on the  $m = 2$  available CPUs. If it decides to schedule only  $\tau_1$ , we obtain state  $\mathcal{T}_2$  after one clock tick. From the game graph in Figure 1, it is easy to see that this is bad choice for the scheduler. Indeed, from  $\mathcal{T}_2$ , player  $\mathcal{T}$  can move the system to  $\mathcal{S}_2$ , and in all successors ( $\mathcal{T}_4, \dots, \mathcal{T}_7$ ) of  $\mathcal{S}_2$  a task misses its deadline (depicted by thick node borders). Hence, whatever the scheduler plays from  $\mathcal{S}_2$  and  $\mathcal{T}_2$ , he loses the game. On the other hand, scheduling  $\{\tau_1, \tau_2\}$  and  $\{\tau_2, \tau_3\}$  from  $\mathcal{S}_1$  and  $\mathcal{S}_3$  respectively, guarantees the scheduler to win the game (all successors of  $\mathcal{T}_{11}$  that are not shown are winning too).

## 2.1 Scheduling games

Let us now formally define scheduling games. We start by the notion of system state that models the current status of all tasks in the system [BC07]. In all states  $S$  of the system, we store two pieces of information for each task  $\tau_i$ : (i) The earliest next arrival time  $\text{NAT}_S(\tau_i)$  of the next job of  $\tau_i$  and (ii) the remaining computing time  $\text{RCT}_S(\tau_i)$  of the current job<sup>2</sup> of  $\tau_i$ .

**Definition 1** (System states). *Let  $\tau = \{\tau_1, \dots, \tau_n\}$  be a set of sporadic tasks. A system state  $S$  of  $\tau$  is a pair  $(\text{NAT}_S, \text{RCT}_S)$  where: (i)  $\text{NAT}_S$  is a function assigning to all tasks  $\tau_i$  a value  $\text{NAT}_S(\tau_i) \leq T_i$ ; and (ii)  $\text{RCT}_S$  is a function assigning to all tasks  $\tau_i$  a value  $\text{RCT}_S(\tau_i) \leq C_i$*

Observe that, since the NAT parameter is not bounded below, there are infinitely many system states. We will limit the number of states to a finite set when defining the game. A task  $\tau_i$  is said to be *active* in state  $S$  if it currently has a job that has not finished in  $S$ , i.e., the set of active tasks in  $S$  is  $\text{ACTIVE}(S) = \{\tau_i \mid \text{RCT}_S(\tau_i) > 0\}$ . To the contrary, we say that  $\tau_i$  is *idle* in  $S$  iff  $\text{RCT}_S(\tau_i) = 0$ . A task  $\tau_i$  is *eligible* in  $S$  if it can submit a job from this configuration, i.e., the set of eligible tasks in  $S$  is:  $\text{ELIGIBLE}(S) = \{\tau_i \mid \text{NAT}_S(\tau_i) \leq 0 \wedge \text{RCT}_S(\tau_i) = 0\}$ . Finally, The *laxity* of  $\tau_i$  in  $S$  is the value  $\text{LAXITY}_S(\tau_i) = \text{NAT}_S(\tau_i) - (T_i - D_i) - \text{RCT}_S(\tau_i)$ . Intuitively, the laxity of a task measures the amount of forthcoming CPU steps that the task could remain idle without taking the risk to miss its deadline. In particular, states where the laxity is negative will be declared as losing states.

Now let us define the possible moves of both players. Let  $S \in \text{STATES}(\tau)$  be a system state. We first define the possible moves of player  $\mathcal{S}$ , i.e. the scheduler. One move of  $\mathcal{S}$  is characterised by the choice of the set  $\tau'$  of tasks that will be scheduled. Formally, for all  $\tau' \subseteq \text{ACTIVE}(S)$  s.t.  $|\tau'| \leq m$  (i.e.,  $\tau'$  does not contain more tasks than the  $m$  available CPUs), we let  $\text{Succ}_{\mathcal{S}}(S, \tau')$  be the (unique) state  $S'$  s.t.: (i)  $\text{NAT}_{S'}(\tau_i) = \text{NAT}_S(\tau_i) - 1$ ; and (ii)  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i)$  if  $\tau_i \notin \tau'$ . Otherwise,  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i) - 1$ . Let us now define the possible moves of player  $\mathcal{T}$ , i.e. the tasks. Each move of  $\mathcal{T}$  is characterised by a set  $\tau'$  of eligible tasks. We let  $\text{Succ}_{\mathcal{T}}(S, \tau') \subseteq \text{STATES}(\tau)$  be the set of system states s.t.  $S' \in \text{Succ}_{\mathcal{T}}(S, \tau')$  iff: (i)  $\text{NAT}_{S'}(\tau_i) = \text{NAT}_S(\tau_i)$  if  $\tau_i \notin \tau'$ . Otherwise,  $\text{NAT}_{S'}(\tau_i) = T_i$ ; (ii) And if  $\tau_i \notin \tau'$ ,  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i)$ . Otherwise,  $\text{RCT}_{S'}(\tau_i) = C_i$ . Finally, we let  $\text{Succ}(S) = \text{Succ}_{\mathcal{S}} \cup \text{Succ}_{\mathcal{T}}$ .

**Example 1.** *The initial state of the game  $G_{\tau, m}$  in our running example (state  $\text{init}^{\mathcal{T}}$  in Figure 1) contains the system state  $\langle (0, 0), (0, 0), (0, 0) \rangle$  (we denote a system state  $S$  by the tuple  $\langle (\text{RCT}(\tau_1), \text{NAT}(\tau_1)), \dots, (\text{RCT}(\tau_n), \text{NAT}(\tau_n)) \rangle$ ). Then,  $\mathcal{S}_1 = \langle (1, 2), (2, 2), (1, 2) \rangle$ . In  $\mathcal{S}_1$  all tasks are eligible,  $\text{LAXITY}_{\mathcal{S}_1}(\tau_1) = 0$  and  $\text{LAXITY}_{\mathcal{S}_1}(\tau_3) = 3$ . Also,  $\mathcal{T}_3 = \langle (0, 1), (1, 1), (1, 1) \rangle$ , which shows that the job of  $\tau_1$  that had been initially submitted has now completed (so,  $\tau_1$  is idle in  $\mathcal{T}_3$ ), but we must still wait one time unit before all tasks can submit a new job (their NAT are all equal to 1). This explains why the only successor of  $\mathcal{T}_3$  is  $\text{Succ}_{\mathcal{T}}(\mathcal{T}_3, \emptyset)$ . Finally,  $\tau_2$  has now missed its deadline in  $\mathcal{T}_{10} = \langle (0, 0), (1, 0), (0, 0) \rangle$ . Indeed  $\text{LAXITY}_{\mathcal{T}_{10}}(\tau_2) < 0$ .*

<sup>2</sup>If several jobs of the same task are active at the same time, they are treated one at a time, in FIFO order.

For a set  $\tau = \{\tau_1, \dots, \tau_n\}$  of sporadic tasks, and  $m$  CPUs, we let  $G_{\tau, m} = \langle V_S, V_{\mathcal{T}}, E, I, \text{Bad} \rangle$  where:

- $V_S = \{S \in \text{STATES}(\tau) \mid \forall 1 \leq i \leq n : \text{LAXITY}_S(\tau_i) \geq -1\} \times \{\mathcal{S}\}$  is the set of scheduler-controlled nodes.
- $V_{\mathcal{T}} = \{S \in \text{STATES}(\tau) \mid \forall 1 \leq i \leq n : \text{LAXITY}_S(\tau_i) \geq -1\} \times \{\mathcal{T}\}$  is the set of nodes controlled by the tasks.
- $E = E_S \cup E_{\mathcal{T}}$  is the set of edges where:
  - $E_S$  is the set of scheduler moves. It contains an edge  $((S, \mathcal{S}), (S', \mathcal{T}))$  iff there is  $\tau' \subseteq \text{ACTIVE}(S)$  s.t.  $|\tau'| \leq m$  and  $S' = \text{Succ}_S(S, \tau')$ . In this case, we sometimes abuse the notations, and consider that this edge is labelled by  $\tau'$ , denoting it  $((S, \mathcal{S}), \tau', (S', \mathcal{T}))$ .
  - $E_{\mathcal{T}}$  is the set of tasks moves. It contains an edge  $((S, \mathcal{T}), (S', \mathcal{S}))$  iff there exists  $\tau' \subseteq \text{ELIGIBLE}(S)$  s.t.  $S' \in \text{Succ}_{\mathcal{T}}(S, \tau')$ . Again, we abuse notations and denote this edge by  $((S, \mathcal{T}), \tau', (S', \mathcal{S}))$ .
- $I = (S_0, \mathcal{T})$ , where for all  $1 \leq i \leq n$ :  $\text{RCT}_{S_0}(\tau_i) = \text{NAT}_{S_0}(\tau_i) = 0$  is the initial state.
- $\text{Bad} = \{(S, \mathcal{T}) \in V_{\mathcal{T}} \mid \exists \tau_i \in \text{ACTIVE}(S) \text{ such that } \text{LAXITY}_S(\tau_i) < 0\}$ , i.e.  $\text{Bad}$  is the set of failure states.

Observe that in this definition, we consider only states where the laxity of all tasks is  $\geq -1$ . It is easy to check (see [GGL13, BC07] for the details) that this restriction guarantees the number of game nodes to be *finite*. Moreover, this is sufficient to detect all deadline misses, as any execution of the system where a task misses a deadline will necessarily traverse a state with a laxity equal to  $-1$  for one of the tasks [GGL13, BC07]. In the sequel we lift the NAT and RCT notations to nodes of the games, i.e., for a node  $v = (S, \mathcal{P})$ , we let  $\text{NAT}_v = \text{NAT}_S$  and  $\text{RCT}_v = \text{RCT}_S$ .

## 2.2 Computation of winning strategies

Now note that the syntax of scheduling games is fixed, let us explain what we want to compute on those games, i.e. *winning strategies* for player  $\mathcal{S}$ . Fix a scheduling game  $G_{\tau, m} = \langle V_S, V_{\mathcal{T}}, E, I, \text{Bad} \rangle$ . First, a *play* in  $G_{\tau, m}$  is a path in the game graph, i.e. a sequence  $v_0, v_1, \dots, v_i, \dots$  of graph vertices (for all  $i \geq 0$ :  $v_i \in V_S \cup V_{\mathcal{T}}$ ) s.t. for all  $i \geq 0$ :  $(v_i, v_{i+1}) \in E$ . Then, a *strategy* for player  $\mathcal{S}$  is a function  $\sigma : V_S \rightarrow V_{\mathcal{T}}$  s.t. for all  $v \in V_S$ :  $(v, \sigma(v)) \in E$ . That is,  $\sigma$  assigns, to each node of  $V_S$ , one of its successor<sup>3</sup> to be played when  $v$  is reached. Given a strategy  $\sigma$ , the *outcome* of  $\sigma$  from a node  $v \in V_S \cup V_{\mathcal{T}}$  is the set  $\text{Outcome}(\sigma, v)$  of all possible plays in  $G_{\tau, m}$  that start in  $v$  and are obtained with  $\mathcal{S}$  playing according to  $\sigma$ , i.e.,  $v_0, v_1, \dots, v_i, \dots \in \text{Outcome}(\sigma, v)$  iff (i)  $v_0 = v$ ; and (ii) for all  $i \geq 0$ :  $v_i \in V_S$  implies  $v_{i+1} = \sigma(v_i)$ . We denote by  $\text{Outcome}(\sigma)$  the set  $\text{Outcome}(\sigma, I)$ . Then, a strategy  $\sigma$  is *winning* from a node  $v$  iff it ensures  $\mathcal{S}$  to

<sup>3</sup>In general, strategies might need to have access to the whole prefix of execution to determine what to play next. However, since scheduling games are a special case of safety games [Tho95], it is well-known that *positional* strategies are sufficient, i.e. strategies whose output depends only on the current game vertex.

always avoid **Bad** (from  $v$ ) whatever  $\mathcal{T}$  does, i.e.,  $\sigma$  is winning from  $v$  iff for all plays  $v_0, v_1, \dots, v_i, \dots \in \text{Outcome}(\sigma, v)$ , for all  $i \geq 0$ :  $v_i \notin \text{Bad}$ . A strategy  $\sigma$  is simply said *winning* if it is winning from the initial node  $I$ . We say that a node  $v$  is *winning* (respectively *losing*) iff there exists a (there is no) strategy that is winning from  $v$ . We denote by **Win** and **Lose** the sets of winning and losing nodes respectively. It is well-known<sup>4</sup> that, in our setting, all nodes are either winning or losing, i.e.,  $\text{Win} \cup \text{Lose} = V_{\mathcal{T}} \cup V_{\mathcal{S}}$ . Hence, computing one of those sets is sufficient to deduce the other.

It is easy to check that, given a set of sporadic tasks  $\tau$  and a number  $m$  of CPUs, the scheduling game  $G_{\tau, m}$  captures all the possible behaviours of  $\tau$  on a platform of  $m$  identical CPUs, running under any possible online scheduler. Then, the answer to the online feasibility problem is positive (i.e., there exists an online scheduler) iff there is a winning strategy in the game, i.e., iff  $I \in \text{Win}$ , or, equivalently, iff  $I \notin \text{Lose}$  (this is the condition that our algorithms will check). Actually, the winning strategy  $\sigma$  is the scheduler itself: by definition, every pair  $(v, \sigma(v))$  corresponds to an edge  $(v, \tau', \sigma(v))$  in  $G_{\tau, m'}$ , which means that, in state  $v$ , the scheduler must grant the CPUs to all tasks in  $\tau'$ . Since  $\sigma$  is winning no bad state will ever be reached, hence, no deadline will be missed.

**Example 2.** *In the example graph of Figure 1, a winning strategy is a strategy  $\sigma$  s.t.  $\sigma(\mathcal{S}_1) = \mathcal{T}_3$  and  $\sigma(\mathcal{S}_3) = \mathcal{T}_{11}$ . It is easy to check that (on this graph excerpt), playing this strategy allows to avoid  $\text{Bad} = \{\mathcal{T}_4, \mathcal{T}_5, \dots, \mathcal{T}_{10}\}$ , whatever choice  $\mathcal{T}$  makes.*

Let us now recall a classical algorithm to solve *safety games*, a broad class of games played on graphs to which scheduling games belong. The algorithm is a variant of *backward induction*: it consists in computing *the set of all nodes from which  $\mathcal{S}$  cannot guarantee to avoid Bad*. This set, denoted **Attr** is called the *attractor* of **Bad**; in Figure 1, it corresponds to the gray nodes. The computation is done inductively, starting from **Bad**, and following the graph edges in a backward fashion (hence the name). Formally, the algorithm relies on the two following operators, for a set  $V \subseteq V_{\mathcal{T}} \cup V_{\mathcal{S}}$ .

$$\exists\text{Pre}(V) = \{v \mid \text{Succ}(v) \cap V \neq \emptyset\} \quad (1)$$

$$\forall\text{Pre}(V) = \{v \mid \text{Succ}(v) \subseteq V\}. \quad (2)$$

That is,  $v \in \exists\text{Pre}(V)$  iff *there exists a successor* of  $v$  which is in  $V$ ; and  $v \in \forall\text{Pre}(V)$  iff *all successors* of  $v$  belong to  $V$ . Then, given  $V \subseteq V_{\mathcal{T}} \cup V_{\mathcal{S}}$ , we let  $\text{UPre}(V)$  be the set of *uncontrollable predecessors* of  $V$ :  $\text{UPre}(V) = \exists\text{Pre}(V \cap V_{\mathcal{S}}) \cup \forall\text{Pre}(V \cap V_{\mathcal{T}})$ . Intuitively,  $v \in \text{UPre}(V)$  iff  $\mathcal{S}$  cannot prevent the game to reach  $V$  (in one step) from  $v$ , either because  $v$  is a  $\mathcal{T}$ -node that has a successor (which is necessarily a  $\mathcal{S}$ -node) in  $V$ , or because  $v$  is an  $\mathcal{S}$ -node that has all its successors (that are all  $\mathcal{T}$ -nodes) in  $V$ . Thus, if  $V$  contains only losing nodes, then all nodes in  $\text{UPre}(V)$  are losing too.

Equipped with these definitions, we can describe now the backward algorithm (Algorithm 1) for solving scheduling games, that will be the basis of our contribution. It computes a sequence  $(\text{Attr}_i)_{i \geq 0}$  of sets of states with the following property:  $v \in \text{Attr}_i$  iff player  $\mathcal{T}$  has a strategy to force the game to reach **Bad** in

<sup>4</sup>This property is called *determinacy* and can be established using the classical result of Donald Martin [Mar75].



---

**Algorithm 1:** Backward algorithm to compute Lose.

---

```

1 ES begin
2    $i \leftarrow 0$ 
3    $\text{Attr}_0 \leftarrow \text{Bad}$ 
4   repeat
5      $\text{Attr}_{i+1} \leftarrow \text{Attr}_i \cup \text{UPre}(\text{Attr}_i)$ 
6      $i \leftarrow i + 1$ 
7   until  $\text{Attr}_i = \text{Attr}_{i-1}$ 
8   return  $\text{Attr}_i$ 

```

---

at most  $i$  steps from  $v$ . Since the sets  $\text{Attr}_i$  (which contain nodes of the game graph) grow along with the iterations of the algorithm, and since the graph is finite, the algorithm necessarily terminates. It is well-known that it computes exactly the set of losing states:

**Theorem 1** (Taken from [Tho95]). *When applied to a scheduling game  $G = \langle V_S, V_T, E, I, \text{Bad} \rangle$ , Algorithm 1 terminates in at most  $|V_T \cup V_S|$  steps and returns the set of losing states Lose.*

Thus, the answer to the online feasibility problem is positive iff  $I$  does not belong to the set returned by the algorithm. When it is the case, a winning strategy (hence also a scheduler) can easily be obtained: in all nodes  $v \in V_S$  that are visited, we let  $\sigma(v)$  be any node  $v'$  s.t.  $(v, v') \in E$  and  $v' \notin \text{Lose}$ . Such a node  $v'$  is guaranteed to exist by construction (otherwise, by definition of  $\forall\text{Pre}$ ,  $v$  would have been added to the attractor, and would not be winning).

**Example 3.** *On the graph in Figure 1, we have:  $\text{Attr}_0 = \{\mathcal{T}_4, \mathcal{T}_5, \dots, \mathcal{T}_{10}\}$ ;  $\text{Attr}_1 = \text{Attr}_0 \cup \{\mathcal{S}_2\}$ ;  $\text{Attr}_2 = \text{Attr}_1 \cup \{\mathcal{T}_2\}$ ; and  $\text{Attr}_2 = \text{Attr}_3$ , so the algorithm converges after 3 steps.*

While this algorithm is a nice theoretical solution, one needs to answer the following questions to obtain a practical implementation: (i) how can we compute a representation of **Bad**? and (ii) given  $\text{Attr}_i$ , how can we compute  $\text{Attr}_{i+1}$ ? Since the game graph is finite, these two questions can be solved by building the whole game graph, and analysing it. However, in practice, the graph has a size which is exponential in the description of the problem instance [BM10], so applying this algorithm straightforwardly is not possible in practice. The core contribution of the present work is to propose heuristics that exploit the particular structure of scheduling games, in order to speed up Algorithm 1.

### 3 Antichain techniques for sporadic tasks

In this section, we introduce the main technical contribution of the paper: an *antichain*-based [DR10] heuristic to speed up the performance of the backward algorithm for solving scheduling games. Let us start with an intuitive discussion of our heuristic. It relies on the definition of a partial order  $\sqsupseteq$  that compares the state of the game graph, with the following important property: *if  $v$  is a losing node, then, all ‘bigger’ nodes  $v'$  (i.e., with  $v' \sqsupseteq v$ ) are also losing*<sup>5</sup>.

<sup>5</sup>Actually, the symmetrical property is also true: if  $v \in \text{Win}$ , then,  $v \sqsupseteq v'$  implies  $v' \in \text{Win}$ .

Then, the optimisation of Algorithm 1 consists, roughly speaking, in *keeping in the sets  $\text{Attr}_i$  the minimal elements only*. This has two consequences. First, since there are, in practice, many comparable elements in the sets  $\text{Attr}_i$ , the memory footprint of the algorithm is dramatically reduced. Second, we perform the computation of the uncontrollable predecessors on the minimal elements only (instead of computing the uncontrollable predecessors of all the elements in  $\text{Attr}_i$ ), which reduces significantly the running time of each iteration of the algorithm. It should now be clear that the definition of  $\sqsupseteq$  partial order is central to our improved algorithm. Its formal definition will be given later, but we can already sketch its intuition through the following example:

**Example 4.** *On the graph given in Figure 1, we have  $\mathcal{T}_4 \sqsupseteq \mathcal{T}_5$ . This means that since  $\mathcal{T}_5$  is losing (it is in **Bad**), then  $\mathcal{T}_4$  should be too (it is indeed the case). So, intuitively,  $\mathcal{T}_4$  is a node which (from the scheduler's point of view) is 'harder' to win from than  $\mathcal{T}_5$ . Indeed, one can check that  $\mathcal{T}_4$  corresponds to system state  $\langle (0, 1), (2, 0), (1, 0) \rangle$  and  $\mathcal{T}_5$  to  $\langle (0, 1), (1, 0), (1, 0) \rangle$ . So,  $\text{RCT}_{\mathcal{T}_4}(\tau_2) > \text{RCT}_{\mathcal{T}_5}(\tau_2)$ , while all the other parameters of the states are equal. It is thus not surprising that  $\mathcal{T}_4$  is 'harder' than  $\mathcal{T}_5$ , since  $\tau_2$  needs more CPU time in  $\mathcal{T}_4$  than in  $\mathcal{T}_5$  with the same time remaining to its deadline. This also means that in our improved algorithm, we will not keep  $\mathcal{T}_4$  in the initial set of states we compute  $\text{Attr}_0$ , since  $\mathcal{T}_4$  is not minimal in **Bad**.*

### 3.1 Partial orders, antichains and closed sets

Let us now introduce the definitions on which our techniques rely. Fix a finite set  $S$ . A relation  $\sqsupseteq \in S \times S$  is a partial order iff  $\sqsupseteq$  is reflexive, transitive and antisymmetric. As usual, we often write  $s \sqsupseteq s'$  and  $s \not\sqsupseteq s'$  instead of  $(s, s') \in \sqsupseteq$  and  $(s, s') \notin \sqsupseteq$ , respectively. The  $\sqsupseteq$ -upward closure  $\uparrow^{\sqsupseteq}(S')$  of a set  $S' \subseteq S$  is the smallest set containing  $S'$  and all the elements that are larger than some element in  $S'$ , i.e.,  $\uparrow^{\sqsupseteq}(S') = \{s \mid \exists s' \in S' : s \sqsupseteq s'\}$ . Then, a set  $S'$  is *upward closed* iff  $S' = \uparrow^{\sqsupseteq}(S')$ . When the partial order is clear from the context, we often write  $\uparrow(S)$  instead of  $\uparrow^{\sqsupseteq}(S)$ . A subset  $\mathcal{A}$  of some set  $S' \subseteq S$  is an *antichain* on  $S'$  with respect to  $\sqsupseteq$  iff it contains only elements that are incomparable wrt  $\sqsupseteq$ , i.e. for all  $s, s' \in \mathcal{A}$ :  $s \neq s'$  implies  $s \not\sqsupseteq s'$ . An antichain  $\mathcal{A}$  on  $S'$  is said to be a set of *minimal elements of  $S'$*  (or a *minimal antichain of  $S'$* ) iff for all  $s_1 \in S'$  there is  $s_2 \in \mathcal{A}$ :  $s_1 \sqsupseteq s_2$ . It is easy to check that if  $\mathcal{A}$  is a minimal antichain of  $S'$ , then  $\uparrow(\mathcal{A}) = \uparrow(S')$ . This is a key observation for our improved algorithm: intuitively,  $\mathcal{A}$  can be regarded as a *compact* representation of  $\uparrow(S')$ , which is of minimal size in the sense that it contains no pair of  $\sqsupseteq$ -comparable elements. Moreover, since  $\sqsupseteq$  is a partial order, each subset  $S'$  of the finite set  $S$  admits a unique minimal antichain, that we denote by  $\lfloor S' \rfloor$ . Observe that one can always effectively build  $\lfloor S' \rfloor$ , simply by iteratively removing from  $S'$ , all the elements that strictly dominate another one.

In our game setting, we will be interested in a particular class of partial orders, which are called *turn-based alternating simulations*:

**Definition 2** ([AHKV98, GGS14]). *Let  $G = (V_S, V_T, E, I, \text{Bad})$  be a safety game. A partial order  $\sqsupseteq \subseteq (V_S \times V_S) \cup (V_T \times V_T)$  is a turn-based alternating simulation relation for  $G$  (tba-simulation for short) iff for all  $v_1, v_2$  s.t.  $v_1 \sqsupseteq v_2$ , either  $v_1 \in \text{Bad}$  or the three following conditions hold: (i) If  $v_1 \in V_S$ , then, for all  $v'_1 \in \text{Succ}(v_1)$ , there is  $v'_2 \in \text{Succ}(v_2)$  s.t.  $v'_1 \sqsupseteq v'_2$ ; (ii) If  $v_1 \in V_T$ , then, for*

all  $v'_2 \in \text{Succ}(v_2)$ , there is  $v'_1 \in \text{Succ}(v_1)$  s.t.  $v'_1 \succeq v'_2$ ; and (iii)  $v_2 \in \text{Bad}$  implies  $v_1 \in \text{Bad}$ .

### 3.2 Improved algorithm

The key observation to our improved algorithm can now be stated formally: the sets  $\text{Attr}_i$  computed by Algorithm 1 are actually *upward-closed* sets (for any partial order which is a tba-simulation), which justifies that we can manipulate them compactly through their minimal elements only:

**Lemma 1.** *Given a set  $V$  which is upward-closed for a tba-simulation  $\succeq$ :  $\text{UPre}(V) = \uparrow^\succeq(\text{UPre}(V))$ .*

*Proof.* We need to prove that  $\text{UPre}(V) \subseteq \uparrow(\text{UPre}(V))$  and  $\uparrow(\text{UPre}(V)) \subseteq \text{UPre}(V)$ . The first item is trivially correct. Now we will prove the second item by showing that  $\forall v \in \uparrow(\text{UPre}(V))$  then  $v \in \text{UPre}(V)$ . Since  $v \in \uparrow(\text{UPre}(V))$ , there hence exists  $v' \in \text{UPre}(V)$  such that  $v \succeq v'$ . We divide into two cases as follows.

- In case  $v, v' \in V \cap V_{\mathcal{T}}$ : By Definition 2, for all  $\bar{v}' \in \text{Succ}(v')$ , there exists  $\bar{v} \in \text{Succ}(v)$  such that  $\bar{v} \succeq \bar{v}'$ . By the definition of  $\forall\text{Pre}(V \cap V_{\mathcal{T}})$  (see equation (2)),  $\bar{v}' \in V$  therefore  $\bar{v} \in V$ . Finally, by the definition of  $\exists\text{Pre}(V \cap V_{\mathcal{S}})$  (see (1)), we derive that  $v \in \exists\text{Pre}(V \cap V_{\mathcal{S}})$  therefore  $v \in \text{UPre}(V)$ .
- In case  $v, v' \in V \cap V_{\mathcal{S}}$ : By Definition 2, for all  $\bar{v} \in \text{Succ}(v)$ , then there exists  $\bar{v}' \in \text{Succ}(v')$  such that  $\bar{v} \succeq \bar{v}'$ . Then since  $v' \in \text{UPre}(V)$  then  $\text{Succ}(v') \subseteq V$  (by the definition of  $\forall\text{Pre}(V \cup V_{\mathcal{T}})$ ). Therefore  $\text{Succ}(v) \subseteq V$ , we derive that  $v \in \forall\text{Pre}(V \cup V_{\mathcal{T}})$  (see equation (2)) and finally  $v \in \text{UPre}(V)$ .

□

**Proposition 1.** *Let  $G$  be a scheduling game and let  $\succeq$  a tba-simulation for  $G$ . Then the sets  $\text{Attr}_i$  computed are upward-closed for  $\succeq$ , i.e. for all  $i \geq 0$ :  $\text{Attr}_i = \uparrow^\succeq(\text{Attr}_i)$ .*

*Proof.* We prove the lemma by induction on  $i$ .

**Base case:** When  $i = 0$ , by the definition,  $\text{Attr}_0 = \text{Bad}$ . We need to prove that  $\text{Bad} = \uparrow(\text{Bad})$ . It is trivial to conclude that  $\text{Bad} \subseteq \uparrow(\text{Bad})$ . It remains to prove that  $\uparrow(\text{Bad}) \subseteq \text{Bad}$ . For each  $v \in \uparrow(\text{Bad})$ , there exists  $v' \in \text{Bad}$  such that  $v \succeq v'$ . Since  $v'$  is a losing state, then  $v$  is losing as well (by the definition of tba-simulation). Hence,  $v \in \text{Bad}$ .

**Inductive case:** For all  $i \geq 1$ , given  $\text{Attr}_i = \uparrow(\text{Attr}_i)$ , we will prove that  $\text{Attr}_{i+1} = \uparrow(\text{Attr}_{i+1})$ . By the definition of the sequences of  $\text{Attr}$ ,  $\text{Attr}_{i+1} = \text{Attr}_i \cup \text{UPre}(\text{Attr}_i)$ . Then we need to prove that:

$$\text{Attr}_i \cup \text{UPre}(\text{Attr}_i) = \uparrow(\text{Attr}_i \cup \text{UPre}(\text{Attr}_i)),$$

i.e.  $\text{UPre}(\text{Attr}_i) = \uparrow(\text{UPre}(\text{Attr}_i))$  that has been done by Lemma 1. □

We can now define the improvement to Algorithm 1, that consists in manipulating the sequence of sets  $(\text{Attr}_i)_{i \geq 0}$  by their minimal elements only. To this end, we define  $\text{UPre}^\#$ , a variant of the  $\text{UPre}$  operator that works directly

on the minimal elements. It receives an antichain of minimal elements  $\mathcal{A}$  (that represents the upward-closed set  $\uparrow(\mathcal{A})$ ) and returns the antichain of minimal elements of  $\text{UPre}(\uparrow(\mathcal{A}))$ :

$$\text{UPre}^\#(\mathcal{A}) = \lfloor \text{UPre}(\uparrow(\mathcal{A})) \rfloor \quad (3)$$

Clearly, since the game graph is finite and since  $\text{UPre}$  can be computed,  $\text{UPre}^\#(\mathcal{A})$  is computable too: it suffices to ‘expand’ the antichain  $\mathcal{A}$  to  $\uparrow(\mathcal{A})$  (which is a finite set), compute  $\text{UPre}(\uparrow(\mathcal{A}))$ , then keep only the minimal elements. However, this procedure deceives the purpose of using a compact representation of upward-closed sets, so we will introduce later an efficient way to compute  $\text{UPre}^\#(\mathcal{A})$  for a given partial order. Equipped with  $\text{UPre}^\#$ , we can now define our improved algorithm, given in Algorithm 2. As announced, it computes a sequence  $(\text{AntiLosing}_i)_{i \geq 0}$  of antichains representing respectively the sets in the  $(\text{Attr}_i)_{i \geq 0}$  sequence, as stated by the following proposition:

---

**Algorithm 2:** Backward traversal with tba-simulation, for the computation of  $\lfloor \text{Lose} \rfloor$

---

```

1 BW-TBA begin
2    $i \leftarrow 0$ 
3    $\text{AntiLosing}_0 \leftarrow \lfloor \text{Bad} \rfloor$ 
4   repeat
5      $\text{AntiLosing}_{i+1} \leftarrow \lfloor \text{AntiLosing}_i \cup \text{UPre}^\#(\text{AntiLosing}_i) \rfloor$ 
6      $i \leftarrow i + 1$ 
7   until  $\text{AntiLosing}_i = \text{AntiLosing}_{i-1}$ 
8   return  $\text{AntiLosing}_i$ 

```

---

**Proposition 2.** For all  $i \in \mathbb{N}$ :  $\text{Attr}_i = \uparrow(\text{AntiLosing}_i)$

*Proof.* The proof is by induction on  $i$ .

**Base case** ( $i = 0$ ): When  $i = 0$ ,  $\text{Attr}_0 = \text{Bad}$  and  $\text{AntiLosing}_0 = \lfloor \text{Bad} \rfloor$ . This case is hence trivially correct.

**Inductive case** ( $i = k$ ): The induction hypothesis is that,  $\text{Attr}_{k-1} = \uparrow(\text{AntiLosing}_{k-1})$ . Let us prove that  $\text{Attr}_k = \uparrow(\text{AntiLosing}_k)$ . By the definitions of the two sequences, it is necessary to prove that

$$\begin{aligned}
& \text{Attr}_{k-1} \cup \text{UPre}(\text{Attr}_{k-1}) \\
&= \uparrow(\text{AntiLosing}_{k-1} \cup \text{UPre}^\#(\text{AntiLosing}_{k-1})) \\
&= \uparrow(\text{AntiLosing}_{k-1}) \cup \uparrow(\text{UPre}^\#(\text{AntiLosing}_{k-1})).
\end{aligned}$$

However:

$$\text{Attr}_{k-1} = \uparrow(\text{AntiLosing}_{k-1}) \quad (4)$$

by induction hypothesis and:

$$\text{UPre}(\text{Attr}_i) = \uparrow(\text{UPre}^\#(\text{AntiLosing}_i)) \quad (5)$$

by definition of  $\text{UPre}^\#$ , see equation (3).  $\square$

Based on this proposition, and on the fact that Algorithm 1 computes the sequence  $(\text{Attr}_i)_{i \geq 0}$  and returns **Lose**, we conclude that Algorithm 2 is correct, in the sense that it computes the minimal elements of **Lose**:

**Theorem 2.** *When applied to a scheduling game  $G$  equipped with a tba-simulation  $\sqsupseteq$ , Algorithm 2 terminates and returns  $\lfloor \text{Lose} \rfloor$ .*

*Proof.* By Theorem 1, we know that there is  $k$  s.t.  $\text{Attr}_k = \text{Attr}_{k+1} = \text{Lose}$ . Hence, by Proposition 2,  $\text{AntiLosing}_k = \lfloor \text{Attr}_k \rfloor = \lfloor \text{Attr}_{k+1} \rfloor = \text{AntiLosing}_{k+1}$ . Hence, Algorithm 2 terminates, and returns a set which is equal to  $\lfloor \text{Attr}_k \rfloor = \lfloor \text{Lose} \rfloor$  (remark that Algorithm 2 could converge earlier than Algorithm 1).  $\square$

## 4 The improved algorithm in practice

Until now, we have proven that given a scheduling game  $G$ , and a tba-simulation  $\sqsupseteq$  for  $G$ , Algorithm 2 terminates and returns the minimal elements of the set of losing states. However, this algorithm is parameterised by the definition of a proper tba-simulation (from which the definition of  $\text{UPre}^\#$  depends too), which is strong enough to ensure that the sets  $(\text{AntiLosing}_i)_{i \geq 0}$  will be much smaller than their  $(\text{Attr}_i)_{i \geq 0}$  counterparts. The purpose of this section is twofold: (i) to define a tba-simulation  $\sqsupseteq$  that exploits adequately the structure of scheduling games, as sketched above; and (ii) to show how to compute efficiently the set  $\lfloor \text{Bad} \rfloor$  and the operator  $\text{UPre}^\#$  (based on  $\sqsupseteq$ ).

For the tba-simulation, we rely on our previous works [GGNS], where we have introduced the *idle-ext task simulation* partial order and proved that it is tba-simulation:

**Definition 3** (Idle-ext task simulation). *Let  $\tau$  be a set of sporadic tasks on a platform of  $m$  processors and let  $G = (V_S, V_T, E, I, \text{Bad})$  be a scheduling game of  $\tau$  on the platform. Then, the idle-ext tasks partial order  $\sqsupseteq \subseteq V_S \times V_S \cup V_T \times V_T$  is a simulation relation for all  $v_1 = (S_1, \mathcal{P})$ ,  $v_2 = (S_2, \mathcal{P}')$ :  $v_1 \sqsupseteq v_2$  iff  $\mathcal{P} = \mathcal{P}'$  and, for all  $\tau_i \in \tau$ , the three following conditions hold: (i)  $\text{RCT}_{S_1}(\tau_i) \geq \text{RCT}_{S_2}(\tau_i)$ ; and (ii)  $\text{RCT}_{S_2}(\tau_i) = 0$  implies  $\text{RCT}_{S_1}(\tau_i) = 0$ ; and (iii)  $\text{NAT}_{S_1}(\tau_i) \leq \text{NAT}_{S_2}(\tau_i)$ .*

**Theorem 3** (Taken from [GGNS]).  $\sqsupseteq$  is a tba-simulation

### 4.1 Computation of $\lfloor \text{Bad} \rfloor$

Given the  $\sqsupseteq$  tba-simulation, let us now show how to compute  $\text{AntiLosing}_0 = \lfloor \text{Attr}_0 \rfloor = \lfloor \text{Bad} \rfloor$  (i.e., the initial set in Algorithm 2), without computing first the whole set **Bad**. Recall that, by definition **Bad** contains only  $\mathcal{T}$  states (this is sufficient since, if a task misses a deadline at a  $\mathcal{S}$  state, then all its predecessors—which are  $\mathcal{T}$  states—are losing too).

We build the set  $\lfloor \text{Bad} \rfloor$  by considering each task separately. Let us first consider a game containing only one task  $\tau_i$ . Let  $v$  be a state from **Bad**. Then,  $\text{LAXITY}_v(\tau_i) = \text{NAT}_v(\tau_i) - T_i + D_i - \text{RCT}_v(\tau_i) < 0$ . Therefore,  $v \in \text{Bad}$  iff  $\text{RCT}_v(\tau_i) \in \{1, \dots, C_i\}$  and  $\text{NAT}_v(\tau_i) \leq T_i - D_i + \text{RCT}_s(\tau_i) - 1$ . We denote by  $\text{Bad}_{\sqsupseteq \tau_i}$  the set:

$$\text{Bad}_{\sqsupseteq \tau_i} = \{v \mid \exists 1 \leq j \leq C_i : \text{NAT}_v(\tau_i) = T_i - D_i + C_i - j \text{ and } \text{RCT}_v(\tau_i) = C_i - (j + 1)\}.$$

It is easy to check (see hereinafter) that, in this game with a single task  $\tau_i$ ,  $\lfloor \text{Bad} \rfloor = \text{Bad}_{\sqsupseteq \tau_i}$ .

Let us consider now a scheduling game on a set  $\tau = \{\tau_1, \dots, \tau_n\}$  of  $n$  tasks. Let  $v$  be a state from **Bad**. Then, there exists at least one task  $\tau_i \in \tau$  such that  $\text{LAXITY}_v(\tau_i) < 0$ . Hence the compact representation of all states that are bad because  $\tau_i$  has missed its deadline is  $\text{Bad}_{\sqsupseteq \tau_i, \tau}$ , where  $v \in \text{Bad}_{\sqsupseteq \tau_i, \tau}$  iff (i) there is  $B \in \text{Bad}_{\sqsupseteq \tau_i}$  s.t.  $\text{NAT}_B(\tau_i) = \text{NAT}_v(\tau_i)$  and  $\text{RCT}_B(\tau_i) = \text{RCT}_v(\tau_i)$ ; and (ii) for all  $j \neq i$ :  $\text{NAT}_v(\tau_j) = T_j$  and  $\text{RCT}_v(\tau_j) \in \{0, 1\}$ . Finally, we claim that the antichain covering all bad states is  $\text{Bad}_{\sqsupseteq} = \cup_{1 \leq i \leq n} \text{Bad}_{\sqsupseteq \tau_i, \tau}$ .

**Lemma 2.**  $\text{Bad}_{\sqsupseteq}$  is an antichain and  $\text{Bad} = \uparrow(\text{Bad}_{\sqsupseteq})$ .

*Proof.* Let us first prove that  $\text{Bad}_{\sqsupseteq}$  is an antichain. Since  $\forall \tau_i, \text{Bad}_{\sqsupseteq \tau_i}$  is an antichain,  $\text{Bad}_{\sqsupseteq \tau_i, \tau}$  is either. Hence if  $\text{Bad}_{\sqsupseteq}$  is not an antichain then there must exist  $v_i \in \text{Bad}_{\sqsupseteq \tau_i, \tau}$  and  $v_k \in \text{Bad}_{\sqsupseteq \tau_k, \tau}$  such that  $v_i$  and  $v_k$  are comparable w.r.t  $\sqsupseteq$ . By the definition above,  $\text{NAT}_{v_i}(\tau_i) = T_i - D_i + C_i - j$ ,  $\text{NAT}_{v_i}(\tau_k) = T_k$  and  $\text{NAT}_{v_k}(\tau_i) = T_i$ ,  $\text{NAT}_{v_k}(\tau_k) = T_k - D_k + C_k - j'$  where  $j, j' \geq 1$ . Since  $C_i \leq D_i$  and  $C_k \leq D_k$ ,  $\text{NAT}_{v_i}(\tau_i) < \text{NAT}_{v_k}(\tau_i)$  and  $\text{NAT}_{v_k}(\tau_k) < \text{NAT}_{v_i}(\tau_k)$ . Then  $v_i$  and  $v_k$  cannot be comparable (see Definition 3).

For the second item, firstly, we will prove that  $\text{Bad} \subseteq \uparrow \text{Bad}_{\sqsupseteq}$ . Given a state  $s' \in \text{Bad}$ , assume that  $\tau_i \in \tau$  misses its deadline in  $s$ . We will prove that there exists a state  $s \in \text{Bad}_{\sqsupseteq}$  such that  $s' \sqsupseteq s$ . (i) For  $\tau_i$ , because  $\text{LAXITY}_{s'}(\tau_i) = \text{NAT}_{s'}(\tau_i) - T_i + D_i - \text{RCT}_{s'}(\tau_i) < 0$  and  $1 < \text{RCT}_{s'}(\tau_i) < C_i$ . Then  $\text{RCT}$  and  $\text{NAT}$  of  $\tau_i$  at  $s$  is computed as follows:  $\text{RCT}_s(\tau_i) = \text{RCT}_{s'}(\tau_i)$  and  $\text{NAT}_s(\tau_i) = T_i - D_i + \text{RCT}_{s'}(\tau_i) - 1$  (i.e.  $(\text{NAT}_s(\tau_i), \text{RCT}_s(\tau_i)) \in \text{Bad}_{\sqsupseteq \tau_i}$ ). It is easy to see that  $(\text{NAT}_{s'}(\tau_i), \text{RCT}_{s'}(\tau_i)) \sqsupseteq (\text{NAT}_s(\tau_i), \text{RCT}_s(\tau_i))$  (ii) For  $\tau'_i$ , if  $\text{RCT}_{s'}(\tau'_i) = 0$  then  $\text{RCT}_s(\tau'_i) = 0$ . And if  $\text{RCT}_{s'}(\tau'_i) > 0$  then  $\text{RCT}_s(\tau'_i) = 1$ . In both the two cases,  $\text{NAT}_s(\tau'_i) = T'_i$ . Finally, we see that  $(\text{NAT}_{s'}(\tau'_i), \text{RCT}_{s'}(\tau'_i)) \sqsupseteq (\text{NAT}_s(\tau'_i), \text{RCT}_s(\tau'_i))$ .

Now let us prove that  $\text{Bad} \supseteq \uparrow \text{Bad}_{\sqsupseteq}$ . Given  $v \in \uparrow \text{Bad}_{\sqsupseteq}$ , then  $\exists \tau_i \in \tau$  s.t.  $\text{NAT}_s(\tau_i) \leq T_i - D_i + C_i - j$  and  $\text{RCT}_s(\tau_i) \geq C_i - (j - 1)$  where  $j = 1..C_i$ . Then  $\text{LAXITY}_s(\tau_i) \leq T_i - D_i + C_i - j - T_i + D_i - C_i - (j + 1)$ . Therefore,  $\text{LAXITY}_s(\tau_i) \leq -1$ . In consequence  $v \in \text{Bad}$ .  $\square$

## 4.2 Computation of $\text{UPre}^\#(\mathcal{A})$

Given this tba-simulation  $\sqsupseteq$ , let us now show how to compute  $\text{UPre}^\#$ . As for the definition of  $\text{UPre}$ , we split the computation of uncontrollable predecessors according to the types of the nodes, and let:

$$\text{UPre}^\#(\mathcal{A}) = \exists \text{Pre}^\#(\mathcal{A} \cap V_S) \cup \forall \text{Pre}^\#(\mathcal{A} \cap V_T),$$

where the specifications for these two new operators are:

$$\begin{aligned} \exists \text{Pre}^\#(\mathcal{A}) &= \lfloor \exists \text{Pre}(\uparrow(\mathcal{A})) \rfloor && \text{for all antichains } \mathcal{A} \subseteq V_S, \\ \forall \text{Pre}^\#(\mathcal{A}) &= \lfloor \forall \text{Pre}(\uparrow(\mathcal{A})) \rfloor && \text{for all antichains } \mathcal{A} \subseteq V_T. \end{aligned}$$

**Computation of  $\exists \text{Pre}^\#$**  The computation of  $\exists \text{Pre}^\#(\mathcal{A})$  for some antichain  $\mathcal{A}$  of  $\mathcal{S}$  nodes turns out to be easy, as shown by the next lemma. Intuitively, it says that, computing  $\exists \text{Pre}(U)$  for some upward-closed set  $U \subseteq V_T$  which is given to

us by its compact representation (minimal antichain)  $\mathcal{A} = \lfloor U \rfloor$ , boils down to computing  $\exists\text{Pre}(\mathcal{A})$  (i.e., computing the predecessors of the minimal elements). After minimisation,  $\lfloor \exists\text{Pre}(\mathcal{A}) \rfloor$  gives us the compact representation of  $\exists\text{Pre}(U)$ . Observe that this Lemma holds for any tba-simulation  $\succeq$ , and not only for  $\sqsupseteq$ :

**Lemma 3.** *For all antichains  $\mathcal{A} \subseteq V_S$ , for all tba-simulations  $\succeq$ :  $\exists\text{Pre}^\#(\mathcal{A}) = \lfloor \exists\text{Pre}(\mathcal{A}) \rfloor$*

*Proof.* To show this lemma, one must show that  $\lfloor \exists\text{Pre}(\uparrow(\mathcal{A})) \rfloor = \lfloor \exists\text{Pre}(\mathcal{A}) \rfloor$ , since by definition of  $\exists\text{Pre}^\#$ ,  $\exists\text{Pre}^\#(\mathcal{A}) = \lfloor \exists\text{Pre}(\uparrow(\mathcal{A})) \rfloor$ . We thus prove that:

$$\exists\text{Pre}(\uparrow(\mathcal{A})) = \uparrow(\exists\text{Pre}(\mathcal{A})) \quad (6)$$

We show the inclusion in both directions.

First,  $\exists\text{Pre}(\uparrow(\mathcal{A})) \subseteq \uparrow(\exists\text{Pre}(\mathcal{A}))$ . For all  $v \in \exists\text{Pre}(\uparrow(\mathcal{A}))$ , there exists  $v'$  s.t.  $v' \in \text{Succ}(v) \cap \uparrow(\mathcal{A})$ , by definition of  $\exists\text{Pre}$ . Then  $v' \in \uparrow(\text{Succ}(v) \cap \mathcal{A})$ . Hence  $\text{Succ}(v) \cap \mathcal{A} \neq \emptyset$ . Therefore  $v \in \exists\text{Pre}(\mathcal{A})$  (see equation (1)), in consequence  $v \in \uparrow(\exists\text{Pre}(\mathcal{A}))$ . Next, let us show that  $\uparrow(\exists\text{Pre}(\mathcal{A})) \subseteq \exists\text{Pre}(\uparrow(\mathcal{A}))$ . For each  $v \in \uparrow(\exists\text{Pre}(\mathcal{A}))$ , there exists  $v'$  s.t.  $v' \in \exists\text{Pre}(\mathcal{A})$  and  $v \succeq v'$ . There hence exists  $\bar{v}'$ , a successor of  $v'$  and  $v' \in \mathcal{A}$  (since  $\text{Succ}(v') \cap \mathcal{A} \neq \emptyset$ ). By Definition 2, there exists  $\bar{v} \in \text{Succ}(v)$  such that  $\bar{v} \succeq \bar{v}'$ . Since  $\bar{v}' \in \mathcal{A}$  then  $\bar{v} \in \uparrow(\mathcal{A})$ . Hence,  $v \in \exists\text{Pre}(\uparrow(\mathcal{A}))$ .  $\square$

It remains to observe that, given any set  $V' \subseteq V_S$ , computing  $\lfloor \exists\text{Pre}(V') \rfloor$  can be done directly (i.e., without the need of building and exploring a large portion of the game graph), by ‘inverting’ the definition of the successor relation, and making sure to keep minimal elements only. More precisely, let  $v = (S, \mathcal{S}) \in V_S$  be a  $\mathcal{S}$  node. Then,  $v' = (S', \mathcal{T}) \in \exists\text{Pre}(\{v\})$  iff the following conditions hold for all  $\tau_i \in \tau$ : (i) if  $\text{RCT}_S(\tau_i) < C_i$ , then  $\text{RCT}_S(\tau_i) = \text{RCT}_{S'}(\tau_i)$  and  $\text{NAT}_S(\tau_i) = \text{NAT}_{S'}(\tau_i)$ ; and (ii) if  $\text{RCT}_S(\tau_i) = C_i$  then either  $\text{RCT}_{S'}(\tau_i) = 0$  and  $\text{NAT}_{S'}(\tau_i) = 0$  or  $\text{RCT}_{S'}(\tau_i) = C_i$  and  $\text{NAT}_{S'}(\tau_i) = \text{NAT}_S(\tau_i)$ . In the end,  $\lfloor \exists\text{Pre}(V') \rfloor = \lfloor \bigcup_{v \in V'} \exists\text{Pre}(\{v\}) \rfloor$ .

**Computation of  $\forall\text{Pre}^\#$**  Let us now show how to compute, in an efficient fashion, the operator  $\forall\text{Pre}^\#$ . Remember that we are given an antichain  $\mathcal{A} \subseteq V_T$  which is a compact representation of some upward-closed set  $\uparrow(\mathcal{A})$ , and that we want to compute  $\lfloor \forall\text{Pre}(\uparrow(\mathcal{A})) \rfloor$ , without needing to compute the full  $\uparrow(\mathcal{A})$  set. Unfortunately, and contrary to  $\exists\text{Pre}^\#$ , it is not sufficient to consider the predecessors of the elements in  $\mathcal{A}$  to deduce  $\forall\text{Pre}^\#(\mathcal{A})$ , as shown by the example in Figure 2. In this example, the antichain  $\mathcal{A}$  is  $\{\mathcal{T}_1, \mathcal{T}_2\}$ . The gray cones depict the upward-closures of those states. Now, if we consider the predecessors of  $\mathcal{T}_1 \in \mathcal{A}$ , we obtain  $\mathcal{S}_1$ , however this state has a successor  $\mathcal{T}_3$  which is not in  $\uparrow(\mathcal{A})$ , so  $\mathcal{S}_1 \notin \forall\text{Pre}(\uparrow(\mathcal{A}))$ , hence  $\mathcal{S}_1 \notin \forall\text{Pre}^\#(\mathcal{A})$ . Yet, on this example, there is a state  $\mathcal{S}'_1 \sqsupseteq \mathcal{S}_1$  which has all its successors in  $\uparrow(\mathcal{A})$ , but which is neither in  $\exists\text{Pre}(\mathcal{A})$  nor in  $\forall\text{Pre}(\mathcal{A})$ .

Unfortunately, we haven’t managed to find a *direct* way to compute  $\forall\text{Pre}^\#(\mathcal{A})$  from  $\mathcal{A}$ ; i.e., without enumerating some elements of  $\uparrow(\mathcal{A})$  that are not in  $\mathcal{A}$ . We propose a method that, while not working only on elements of  $\mathcal{A}$ , avoids the full enumeration of  $\uparrow(\mathcal{A})$ , and performs well in practice. It is described in Algorithm 3. Instead of computing  $\forall\text{Pre}^\#(\mathcal{A})$ , it returns the subset  $\forall\text{Pre}^\#(\mathcal{A}) \setminus \uparrow(\mathcal{A})$ . Observe

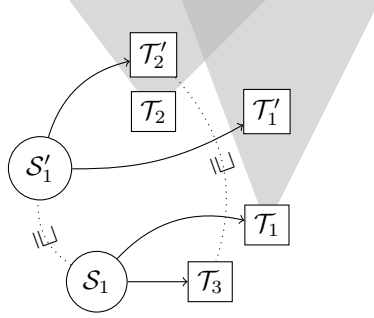


Figure 2:  $\forall\text{Pre}^\#(\{\mathcal{T}_1, \mathcal{T}_2\})$  cannot be computed by considering only the predecessors of  $\mathcal{T}_1$  and  $\mathcal{T}_2$ .

that this subset is sufficient for Algorithm 2, because, at each step, one computes, from  $\text{AntiLosing}_i$ , the set  $\text{AntiLosing}_{i+1}$  as:

$$\text{AntiLosing}_{i+1} = [\exists\text{Pre}^\#(\text{AntiLosing}_i) \cup \forall\text{Pre}^\#(\text{AntiLosing}_i) \cup \text{AntiLosing}_i]. \quad (7)$$

---

**Algorithm 3:** Computing  $\forall\text{Pre}^\#(\mathcal{A}) \setminus \uparrow(\mathcal{A})$ .

---

```

1 Compute- $\forall\text{Pre}$  begin
2    $\mathcal{B} \leftarrow \emptyset$ 
3    $\text{ToSearch} \leftarrow \emptyset$ 
4   foreach  $v \in \exists\text{Pre}(\mathcal{A}) \setminus \uparrow(\mathcal{A})$  do
5     if  $\text{Succ}(v) \subseteq \uparrow(\mathcal{A})$  then  $\mathcal{B} \leftarrow [\mathcal{B} \cup \{v\}]$ 
6     else  $\text{ToSearch} \leftarrow \text{ToSearch} \cup \uparrow(v) \setminus \uparrow(\mathcal{A})$ 
7   while  $\text{ToSearch} \neq \emptyset$  do
8     Pick and remove  $v$  from  $\text{ToSearch}$ 
9     if  $\text{Succ}(v) \subseteq \uparrow(\mathcal{A})$  then  $\mathcal{B} \leftarrow [\mathcal{B} \cup \{v\}]$ 
10    else  $\text{ToSearch} \leftarrow \text{ToSearch} \setminus \{v' \mid v \sqsupseteq v'\}$ 
11  return  $\mathcal{B}$ 

```

---

Hence, it is easy to check that replacing  $\forall\text{Pre}^\#(\text{AntiLosing}_i)$  in (7) by:

$$\forall\text{Pre}^\#(\text{AntiLosing}_i) \setminus \uparrow(\text{AntiLosing}_i)$$

does not change the value of the expression. Yet, computing  $\forall\text{Pre}^\#(\mathcal{A}) \setminus \uparrow(\mathcal{A})$  instead of  $\forall\text{Pre}^\#(\mathcal{A})$  allows us to avoid considering elements which are already computed (in  $\mathcal{A}$ ), which makes Algorithm 3 efficient in practice. This algorithm works as follows. First, we build a set  $\text{ToSearch}$  of candidates nodes for the set  $\forall\text{Pre}^\#(\mathcal{A})$ , then we explore it.  $\text{ToSearch}$  is initialised by the loop starting at line 4. In this loop, we consider all the predecessors  $v$  of  $\mathcal{A}$  (we will explain hereinafter how these predecessors can be computed efficiently). We keep only the predecessors  $v$  that are not covered by  $\mathcal{A}$ , and check whether all successors of  $v$  are in  $\uparrow(\mathcal{A})$ . If yes,  $v \in \forall\text{Pre}(\uparrow(\mathcal{A}))$ , and we add  $v$  to the antichain  $\mathcal{B}$  (that will eventually be returned). Remark that the test  $\text{Succ}(v) \subseteq \uparrow(\mathcal{A})$  can be performed



without computing explicitly  $\uparrow(\mathcal{A})$ , by checking that, for all  $v' \in \text{Succ}(v)$ , there is  $v'' \in \mathcal{A}$  s.t.  $v' \sqsupseteq v''$ . If there are some successors of  $v$  that are not in  $\uparrow(\mathcal{A})$ , we must consider all nodes larger than  $v$  as candidates for  $\forall\text{Pre}^\#(\mathcal{A})$ , but not those that are already in  $\uparrow(\mathcal{A})$ , hence, we add to ToSearch the set  $\uparrow(v) \setminus \uparrow(\mathcal{A})$ . The fact that we add  $\uparrow(v) \setminus \uparrow(\mathcal{A})$  instead of  $\uparrow(v)$  is important to keep ToSearch small, which is a key optimisation of the algorithm. Then, once ToSearch has been built, we examine all the elements  $v$  it contains, and check whether they belong to  $\forall\text{Pre}(\uparrow(\mathcal{A}))$ , in the loop starting in line 7. If  $v \in \forall\text{Pre}(\uparrow(\mathcal{A}))$  (line 9),  $v$  is added to  $\mathcal{B}$ . Otherwise, another optimisation of the algorithm occurs:  $v$  is discarded from ToSearch, as well as *all the nodes that are smaller than  $v$*  (line 10). This is correct by properties of the tba-simulation (see [GGS14]), and allows the algorithm to eliminate candidates from ToSearch much quicker.

To conclude the description of Algorithm 3, let us explain how to compute  $\exists\text{Pre}(\mathcal{A})$  efficiently, when  $\mathcal{A} \subseteq V_{\mathcal{T}}$ , as needed in line 4. Let  $(S, \mathcal{T})$  be a  $\mathcal{T}$ -node. Then,  $(S', S) \in \exists\text{Pre}(S, \mathcal{T})$  iff there is a set  $\tau' \subseteq \tau$  of scheduled tasks s.t.  $|\tau'| \leq m$  (at most  $m$  tasks have been scheduled) and the following conditions hold for all  $\tau_i \in \tau$ : (i)  $\text{NAT}_{S'}(\tau_i) = \min(\text{NAT}_S(\tau_i) + 1, T_i)$  (ii)  $\text{RCT}_S(\tau_i) = C_i$  implies  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i)$  (iii)  $\text{RCT}_S(\tau_i) < C_i$  implies  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i)$  if  $\tau_i \notin \tau'$ ; and  $\text{RCT}_{S'}(\tau_i) = \text{RCT}_S(\tau_i) + 1$  otherwise.

### 4.3 Further optimisation of the algorithm

We close our discussion of BW-TBA by describing the last optimisation that further enhances the performance of the algorithm. In Algorithm 2, the  $\text{UPre}^\#$  operator is applied, at each step of the main loop, to the whole  $\text{AntiLosing}_i$  set in order to compute  $\text{AntiLosing}_{i+1}$ . While this makes the discussion of the algorithm easier, it incurs unnecessary computations: instead of computing the uncontrollable predecessors of *all* states in  $\text{AntiLosing}_i$ , one should instead compute the uncontrollable predecessors of the node that have been inserted in  $\text{AntiLosing}_i$  at the previous iteration of the loop *only*. As an example, on the game in Figure 1,  $\text{AntiLosing}_0 = \lfloor \{\mathcal{T}_4, \dots, \mathcal{T}_{10}\} \rfloor$ ;  $\text{AntiLosing}_1 = \text{AntiLosing}_0 \cup \{\mathcal{S}_2\}$  since  $\mathcal{S}_2$  is incomparable to  $\mathcal{T}_4, \dots, \mathcal{T}_{10}$ . Then, to discover that  $\mathcal{T}_2$  is a losing node too, we should only apply  $\text{UPre}^\#$  to  $\mathcal{S}_2$ , and not to  $\mathcal{T}_4, \dots, \mathcal{T}_{10}$  again.

To formalise this idea, we introduce the notion of *frontier*. Intuitively, at each step  $i$  of the algorithm,  $\text{Frontier}_i$  is an antichain that represents the *newly discovered* losing states, i.e. the states that have been declared losing at step  $i$  but were not known to be losing at step  $i - 1$ . To define formally the sequence  $(\text{Frontier}_i)_{i \geq 0}$ , we introduce the operator  $\forall\text{Pre}^*(\mathcal{A}, \mathcal{C})$  defined as  $\forall\text{Pre}^*(\mathcal{A}, \mathcal{C}) = \uparrow(\exists\text{Pre}(\mathcal{C})) \cap \forall\text{Pre}^\#(\mathcal{A})$ . That is, nodes in  $\forall\text{Pre}^*(\mathcal{A}, \mathcal{C})$  have all their successors in  $\mathcal{A}$  and are larger than some predecessor of  $\mathcal{C}$ . Hence,  $\forall\text{Pre}^*(\mathcal{A}, \mathcal{C})$  can be computed by adapting Algorithm 3, substituting  $\exists\text{Pre}(\mathcal{C})$  for  $\exists\text{Pre}(\mathcal{A})$  in line 4. Clearly, if we manage to keep the size of  $\mathcal{C}$  as small as possible, we will further decrease the size of ToSearch, and Algorithm 3 will be even more efficient.

Now let us define the sequence of  $\text{Frontier}_i$  sets. We let  $\text{Frontier}_0 = \lfloor \text{Bad} \rfloor = \text{Bad}_\sqsubseteq$ . For all  $i > 0$ , we let:

$$\text{Frontier}_{i+1} = \lfloor \text{UPre}^*(\text{AntiLosing}_i, \text{Frontier}_i) \rfloor,$$

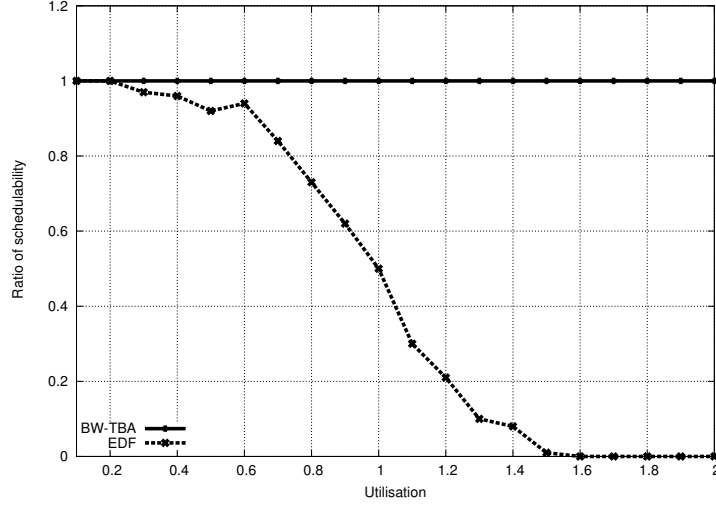


Figure 3: Ratio of schedulable systems by EDF and by BW-TBA.

and:

$$\begin{aligned}
 & \text{UPre}^*(\text{AntiLosing}_i, \text{Frontier}_i) \\
 &= \\
 & \exists \text{Pre}(\text{Frontier}_i \cap V_S) \cup \forall \text{Pre}^*(\text{AntiLosing}_i, \text{Frontier}_i \cap V_T).
 \end{aligned}$$

Observe that in this new definition of the uncontrollable predecessor operator, only predecessors of nodes in the frontier are considered for computation, which is the key of the optimisation. The next lemma and proposition justify the correctness of this new construction. In particular, Proposition 3 shows that the sequence  $(\text{AntiLosing}_i)_{i \geq 0}$  (which is computed by BW-TBA) can be computed applying the operator we have just described, on elements from the frontier only.

To establish Proposition 3, we first need an ancillary lemma, showing that all the elements that occur in the frontier at some step  $i$  are also in  $\text{AntiLosing}_i$ :

**Lemma 4.** *For all  $i \in \mathbb{N}$ ,  $\text{Frontier}_i \subseteq \text{AntiLosing}_i$ .*

*Proof.* We prove the lemma by induction on  $i$ .

**Base case:** When  $i = 0$ ,  $\text{AntiLosing}_0 = \text{Bad}_\sqsupseteq$  and  $\text{Frontier}_0 = \text{Bad}_\sqsupseteq$ . Hence the lemma is trivially correct.

**Inductive case:** Given  $\text{Frontier}_i \subseteq \text{AntiLosing}_i$ , we will prove that:

$$\text{Frontier}_{i+1} \subseteq \text{AntiLosing}_{i+1},$$

i.e.  $\forall v \in \text{Frontier}_{i+1} : v \in \text{AntiLosing}_{i+1}$ .

By the definition of the sequence  $\text{Frontier}$ :

$$\begin{aligned}
 \text{Frontier}_{i+1} &= \lfloor \text{UPre}^*(\text{AntiLosing}_i, \text{Frontier}_i) \rfloor \\
 &= \lfloor \exists \text{Pre}(\text{Frontier}_i \cap V_S) \cup \forall \text{Pre}^*(\text{AntiLosing}_i, \text{Frontier}_i \cap V_T) \rfloor.
 \end{aligned}$$

By the definition of the sequence  $\text{AntiLosing}$ ,  $\text{AntiLosing}_{i+1} = \lfloor \text{AntiLosing}_i \cup \exists \text{Pre}(\text{AntiLosing}_i \cap V_S) \cup \forall \text{Pre}^\#(\text{AntiLosing}_i \cap V_T) \rfloor$ . Hence, proving that the two following items are correct is sufficient.

- (i)  $\exists \text{Pre}(\text{Frontier}_i \cap V_S) \subseteq \exists \text{Pre}(\text{AntiLosing}_i \cap V_S)$
- (ii)  $\forall \text{Pre}^*(\text{AntiLosing}_i, \text{Frontier}_i \cap V_T) \subseteq \forall \text{Pre}^\#(\text{AntiLosing}_i \cap V_T)$ .

We then prove these two items as follows:

- (i) Given  $v \in \exists \text{Pre}(\text{Frontier}_i \cap V_S)$ , then  $\exists v' \in \text{Succ}(v)$  where  $v' \in \text{Frontier}_i \cap V_S$  (see equation (1)). Since  $\text{Frontier}_i \subseteq \text{AntiLosing}_i$  then  $v' \in \text{AntiLosing}_i$  as well. Hence,  $v \in \exists \text{Pre}(\text{AntiLosing}_i \cap V_S)$ .
- (i) Given  $v \in \forall \text{Pre}^*(\text{AntiLosing}_i, \text{Frontier}_i \cap V_T)$  then  $v \in \exists \text{Pre}(\text{Frontier}_i \cap V_T) \cap \forall \text{Pre}^\#(\text{AntiLosing}_i \cap V_T)$ . Therefore,  $v \in \forall \text{Pre}^\#(\text{AntiLosing}_i \cap V_T)$  as well.

Hence the lemma is successfully proved.  $\square$

We can now prove Proposition 3:

**Proposition 3.** *For all  $i \in \mathbb{N}$ :*

$$\text{AntiLosing}_{i+1} = \lfloor \text{AntiLosing}_i \cup \text{Frontier}_{i+1} \rfloor.$$

*Proof of Proposition 3.* The proof is proved by induction on  $i$ .

**Base case:** When  $i = 0$ ,  $\text{AntiLosing}_0 = \text{Frontier}_0 = \text{Bad}_\square$ . By definition of the sequence  $\text{AntiLosing}$ :

$$\begin{aligned} \text{AntiLosing}_1 &= \lfloor \text{AntiLosing}_0 \cup \text{UPre}^\#(\text{AntiLosing}_0) \rfloor \\ \text{UPre}^\#(\text{AntiLosing}_0) &= \exists \text{Pre}(\text{AntiLosing}_0 \cap V_S) \cup \forall \text{Pre}^\#(\text{AntiLosing}_0 \cap V_T). \end{aligned}$$

On the other hand:

$$\begin{aligned} \text{Frontier}_1 &= \lfloor \text{UPre}^*(\text{Frontier}_0, \text{Frontier}_0) \rfloor \\ &= \lfloor \exists \text{Pre}(\text{Frontier}_0 \cap V_S) \cup \forall \text{Pre}^*(\text{AntiLosing}_0, \text{Frontier}_0 \cap V_T) \rfloor. \end{aligned}$$

Since  $\text{AntiLosing}_0 = \text{Frontier}_0$ , then:

$$\forall \text{Pre}^\#(\text{AntiLosing}_0) = \forall \text{Pre}^*(\text{AntiLosing}_0, \text{Frontier}_0).$$

Hence,  $\text{UPre}^\#(\text{AntiLosing}_0) = \text{Frontier}_1$ . Finally,  $\text{AntiLosing}_1 = \lfloor \text{AntiLosing}_0 \cup \text{Frontier}_1 \rfloor$ .

**Inductive case:** For all  $i > 0$ , given  $\text{AntiLosing}_{i+1} = \lfloor \text{AntiLosing}_i \cup \text{Frontier}_{i+1} \rfloor$ , we need to prove  $\text{AntiLosing}_{i+2} = \lfloor \text{AntiLosing}_{i+1} \cup \text{Frontier}_{i+2} \rfloor$ . In other words we need to prove that:

$$\begin{aligned} &\lfloor \text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{AntiLosing}_{i+1} \cap V_S) \cup \forall \text{Pre}^\#(\text{AntiLosing}_{i+1} \cap V_T) \rfloor \\ &= \\ &\left\lfloor \begin{array}{c} \text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{Frontier}_{i+1} \cap V_S) \\ \cup \\ \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1} \cap V_T) \end{array} \right\rfloor. \end{aligned}$$

We divide into two cases as follows.

1. For all states controlled by player  $\mathcal{S}$ , the proposition becomes:

$$\begin{aligned} & \lfloor \text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{AntiLosing}_{i+1}) \rfloor \\ &= \\ & \lfloor \text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{Frontier}_{i+1}) \rfloor. \end{aligned}$$

We need to prove that  $\text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{AntiLosing}_{i+1})$  (abbreviated by  $\mathcal{L}$ ) =  $\text{AntiLosing}_{i+1} \cup \exists \text{Pre}(\text{Frontier}_{i+1})$  (abbreviated by  $\mathcal{R}$ ).

- Assume that  $\exists v \in \mathcal{L}$  such that  $v \notin \mathcal{R}$  ( $v \in V_{\mathcal{T}}$ ). Hence: (i)  $v \notin \text{AntiLosing}_{i+1}$ ; and (ii)  $v \in \exists \text{Pre}(\text{AntiLosing}_{i+1})$ . From 1i, there does not exist  $v' \in \text{AntiLosing}_i$  such that  $v \in \exists \text{Pre}(\{v'\})$ . From 1ii, there exists  $v' \in \text{AntiLosing}_{i+1}$  such that  $v \in \exists \text{Pre}(\{v'\})$ . Hence  $v' \in \text{AntiLosing}_{i+1} \setminus \text{AntiLosing}_i$ , i.e.  $v' \in \text{Frontier}_{i+1}$  (thanks to the hypothesis). In consequence,  $v \in \exists \text{Pre}(\text{Frontier}_{i+1})$ . In contradiction. Therefore  $\mathcal{L} \subseteq \mathcal{R}$ .
- Given  $v \in \mathcal{R}$ , if  $v \in \text{AntiLosing}_{i+1}$  then  $v \in \mathcal{L}$ . Otherwise,  $v \in \exists \text{Pre}(\text{Frontier}_{i+1})$ . Hence there exists  $v' \in \text{Frontier}_{i+1}$  such that  $v \in \exists \text{Pre}(\{v'\})$ . Since  $\text{Frontier}_{i+1} \subseteq \text{AntiLosing}_{i+1}$  (see Lemma 4),  $v' \in \text{AntiLosing}_{i+1}$ . Thus by equation (1),  $v \in \exists \text{Pre}(\text{AntiLosing}_{i+1})$ , thus  $v \in \mathcal{L}$ . Hence,  $\mathcal{L} \supseteq \mathcal{R}$ .

2. For all states controlled by player  $\mathcal{T}$ , the proposition becomes:

$$\begin{aligned} & \lfloor \text{AntiLosing}_{i+1} \cup \forall \text{Pre}^{\#}(\text{AntiLosing}_{i+1}) \rfloor \\ &= \\ & \lfloor \text{AntiLosing}_{i+1} \cup \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1}) \rfloor. \end{aligned}$$

We need to prove  $\text{AntiLosing}_{i+1} \cup \forall \text{Pre}^{\#}(\text{AntiLosing}_{i+1}) = \text{AntiLosing}_{i+1} \cup \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1})$ . For the sake of clarity let us denote:

$$\begin{aligned} \mathcal{L} &= \text{AntiLosing}_{i+1} \cup \forall \text{Pre}^{\#}(\text{AntiLosing}_{i+1}) \\ \mathcal{R} &= \text{AntiLosing}_{i+1} \cup \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1}). \end{aligned}$$

Hence, we need to show that  $\mathcal{L} = \mathcal{R}$ .

- Assume that there exists a state  $v \in \mathcal{L}$  such that  $v \notin \mathcal{R}$ . Therefore  $v \notin \text{AntiLosing}_{i+1}$  (there exists  $v' \notin \text{AntiLosing}_i$  where  $v \in \uparrow(\exists \text{Pre}(v'))$  (a)) and  $v \in \forall \text{Pre}^{\#}(\text{AntiLosing}_{i+1})$ , therefore  $\text{Succ}(v) \subseteq \uparrow(\text{AntiLosing}_{i+1})$  (b), i.e.  $v' \in \text{AntiLosing}_{i+1}$ . Thanks to the hypothesis,  $v' \in \text{AntiLosing}_{i+1} \setminus \text{AntiLosing}_i$ , i.e.  $v' \in \text{Frontier}_{i+1}$  (c). By combining (a), (b) and (c),  $v \in \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1})$ . In consequence  $v \in \mathcal{R}$ . In contradiction. Hence,  $\mathcal{L} \subseteq \mathcal{R}$ .
- Given  $v \in \mathcal{R}$ , if  $v \in \text{AntiLosing}_{i+1}$  then  $v \in \mathcal{L}$ . Otherwise,  $v \in \forall \text{Pre}^*(\text{AntiLosing}_{i+1}, \text{Frontier}_{i+1})$ , therefore  $v \in \forall \text{Pre}^{\#}(\text{AntiLosing}_{i+1})$  by definition of the operator  $\forall \text{Pre}^*(\mathcal{A}, \mathcal{B})$ . Hence,  $\mathcal{L} \supseteq \mathcal{R}$ .

We conclude that  $\text{AntiLosing}_{i+1} = \lfloor \text{AntiLosing}_i \cup \text{Frontier}_{i+1} \rfloor$ .  $\square$

Hence, the formula in line 5 of Algorithm 2 can be replaced by  $\text{AntiLosing}_{i+1} = \lfloor \text{AntiLosing}_i \cup \text{Frontier}_{i+1} \rfloor$  (see Proposition 3), where  $\text{Frontier}_{i+1}$  is computed from  $\text{Frontier}_i$ :  $\text{Frontier}_{i+1} = \lfloor \forall \text{Pre}^*(\text{AntiLosing}_i, \text{Frontier}_i \cap V_{\mathcal{T}}) \cup \text{AntiLosing}_i \cup \exists \text{Pre}(\text{Frontier}_i \cap V_{\mathcal{S}}) \rfloor$ . Since all the computations are now performed on the frontier instead of the whole  $\text{AntiLosing}_i$  set, the performance of Algorithm 2 is improved. This is the algorithm we have implemented and that we report upon in the next section.

## 5 Experimental Results

Let us now report on a series of experiments that demonstrate the practical interest of our approach. Remember that our algorithm performs an *exact test*, i.e. given any real-time system of sporadic tasks, the algorithm always computes an online scheduler if one exists. Otherwise, the algorithm proves the absence of scheduler for the system.

Our implementations are made in C++ using the STL. We performed our experiments on a Mac Pro (mid 2010) server with OS X Yosemite, processor of 3.33 GHz, 6 core Intel Xeon and memory of 32 GB 1333 MHz DDR3 ECC. Our programs were compiled with Apple Inc.’s distribution of g++ version 4.2.1.

We compare the performance of our improved algorithm BW-TBA (Algorithm 2, implemented with the frontier optimisation described in Section 4.3) to three other approaches. The first one consists in scheduling the system with the classical EDF (Earliest Deadline First) scheduler. The second (called ES) consists in first building the portion of the game graph that contains all states that are reachable from  $I$  (the initial state), then applying Algorithm 1 to compute  $\text{Lose}$ , and finally testing whether  $I \in \text{Lose}$  or not. The third approach, called OTFUR-TBA has been introduced in [GGNS]. Contrary to BW-TBA and ES that are *backward approaches* (since we unfold the successor relation in a backward fashion, starting from the Bad states), OTFUR-TBA is a *forward algorithm*. It builds a portion of the game graph and looks for a winning strategy using on-the-fly computation. The idea has originally been introduced as a general algorithm called OTFUR for safety games [CDF<sup>+</sup>05], and OTFUR-TBA is essentially an improved version of OTFUR using antichain techniques (and the  $\sqsubseteq$  partial order), just as BW-TBA is an improved version of ES. In [GGNS], we report on experiments showing OTFUR-TBA outperforms ES in practice.

EDF is known to be *optimal* for arbitrary collections of independent jobs scheduled upon *uniprocessor* platforms. This optimality result, however, is no longer true on *multiprocessor* platforms (that is, there are systems of sporadic tasks for which an online scheduler exists, but that will not be scheduled by EDF). The goal of our first experiment is to compare the number of systems that EDF and BW-TBA can schedule. To this aim, we have generated 2,000 sets of tasks, grouped by values of utilisation  $U$ . For all instances, we consider  $m = 2$  CPUs and  $n = 4$  tasks. For each  $U \in [0.1, 2]$  with a step of 0.1, 100 instances are randomly generated. The  $T$  parameters of all tasks range in the interval  $[5, 15]$ . The generation of this benchmark is based on the UUNIFAST algorithm [BB05].

Figure 3 depicts the ratio of systems scheduled by EDF and BW-TBA on the benchmark described above. All of the 2,000 randomly generated instances are feasible, hence BW-TBA always computes a scheduler, while the ratio of systems that are schedulable by EDF decreases sharply with the increase in the utilisation

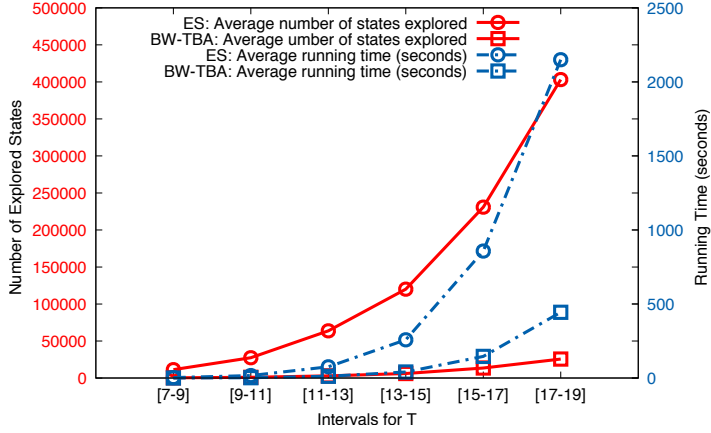


Figure 4: Comparison of the average space explored and the running time between ES vs. BW-TBA for each  $\tau^{[i, i+2]}$ .

factor. This is a first strong point in favour of our approach.

Like BW-TBA, the ES algorithm performs an exact feasibility test, but we naively builds the whole graph of the game which is of exponential size. In order to compare the performance of ES and BW-TBA, we reuse the benchmark presented in [GGNS] (also based on UUNIFAST). It consists of 2,100 sets of tasks grouped by values of  $T$ , the interarrival time. For all instances, we consider  $m = 2$  CPUs and  $n = 3$  tasks. For each  $i$  in  $\{7, 9, 11, 13, 15, 17, 19\}$ , a set called  $\tau^{[i, i+2]}$  of 300 tasks is generated, with a  $T$  parameter in the  $[i, i+2]$  range. For each game instance in the benchmark, we run both BW-TBA and ES and collect two metrics: the number of explored states and the running time.

Figure 4 displays, for each set  $\tau^{[i, i+2]}$ , the average number of states explored and the average running time, of the two algorithms. Our improved algorithm BW-TBA outperforms ES both in terms of space and running time, by approximately one order of magnitude. These results definitely demonstrate the practical interest of antichain techniques for scheduling games.

Now, let us compare the performance of the forward and backward algorithms both enhanced with antichains. We performed another experiment on a benchmark where we let the number of tasks vary. We randomly generated 45 instances of systems with  $n$  tasks for  $n \in \{3, 4, 5, 6\}$  (on a 2 CPUs platform). In all instances, we have  $T \in [5, 7]$ ,  $U \in \{1, 1.5, 2\}$  and  $D \in [C, T]$ . Like the two previous benchmarks, this one is based on UUNIFAST too.

Ratio	$n = 3$	$n = 4$	$n = 5$	$n = 6$
(0, 10%)	0%	4.44%	46.67%	64.44%
[10%, 50%)	88.88%	95.56%	53.33%	35.56%
(50%, 100%)	11.12%	0 %	0 %	0%

Table 2: Percentage of systems with  $n$  tasks for which the ratio on the number of explored nodes (BW-TBA/OTFUR-TBA) falls in the (0, 10%), [10%, 50%) or (50%, 100%) interval.

Our experiments show that, on all instances, BW-TBA explores less states than OTFUR-TBA. This phenomenon happens because BW-TBA computes only the losing states while OTFUR-TBA explores both losing and winning states during search. Table 2, presents the ratio of space explored by BW-TBA and OTFUR-TBA, for example, on systems with 6 tasks, BW-TBA consumes less than 10% of the memory needed by OTFUR-TBA on 64.64% of instances. Note that the memory performance of BW-TBA (vs OTFUR-TBA) improve when the number of tasks increase.

Num. tasks	$n = 3$	$n = 4$	$n = 5$	$n = 6$
Ratio	84.44%	64.44%	44.44%	37.77%

Table 3: The ratio on the number of systems for which BW-TBA runs faster than OTFUR-TBA.

Table 3 presents the ratio of instances on which BW-TBA runs faster than OTFUR-TBA. Notice that the time taken for each instance (of a fixed number of tasks) is variable. For example, on 6-task systems, some are done within one minute; while others take hours to complete. The time taken depends on several hardly predictable characteristics of the system like the number of losing states. These experiments show no clear winner between BW-TBA and OTFUR-TBA, which are rather complementary approaches. In terms of memory, BW-TBA outperforms OTFUR-TBA but OTFUR-TBA might run faster on some systems.

## 6 Conclusion

Inspired by Bonifaci and Marchetti-Spaccamela, we have considered a game-based model for the online feasibility problem of sporadic tasks (on multiprocessor platforms), and we have developed an efficient algorithm to analyse these games, which takes into account the specific structure of the problem. Broadly speaking, we believe that the application of well-chosen formal methods techniques (together with specialised heuristics that exploit the structure of the problems), is a research direction that has a lot of potential and that should be further followed.

## References

- [AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.
- [BB05] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.
- [BC07] Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *Proceedings of OPODIS'07*, volume 4878 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2007.

- [BCM<sup>+</sup>92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.
- [BM10] Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. In *Proceedings of ESA'10*, volume 6347 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2010.
- [CDF<sup>+</sup>05] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of CONCUR'05*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.
- [DB11] Robert Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35, 2011.
- [DR10] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proceedings of TACAS'10*, volume 6015 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2010.
- [EY15] P. Ekberg and Wang Yi. Uniprocessor feasibility of sporadic tasks with constrained deadlines is strongly comp-complete. In *Proceedings of ECRTS'15*, pages 281–286. IEEE, 2015.
- [FGB10] Nathan Fisher, Joël Goossens, and Sanjoy K. Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010.
- [GGL13] Gilles Geeraerts, Joël Goossens, and Markus Lindström. Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm. *Real-Time Systems*, 49(2):171–218, 2013.
- [GGNS] G. Geeraerts, J. Goossens, T. Nguyen, and A. Stainer. Synthesising succinct strategies in safety games with an application to real-time scheduling. Submitted. <http://www.ulb.ac.be/di/verif/ggeeraer/papers/reach-synth.pdf>.
- [GGS14] Gilles Geeraerts, Joël Goossens, and Amélie Stainer. Synthesising succinct strategies in safety and reachability games. In *Proceedings of RP'14*, volume 8762 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2014.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
- [Mar75] Donald A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.
- [PR89] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of ICALP'89*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer, 1989.



- [Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *Proceedings of STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.